Lecture 2:

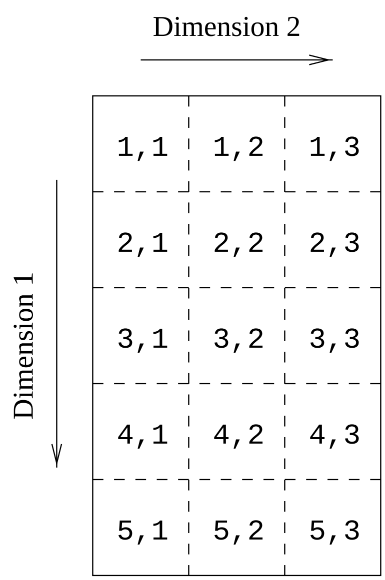
Arrays

Arrays

Arrays (or matrices) hold a collection of different values at the same time. Individual elements are accessed by **subscripting** the array.

A 15 element array can be visualised as:

And a 5 $\times$ 3 array as:



Every array has a type and each element holds a value of that type.

Array Terminology

Examples of declarations:

```
REAL, DIMENSION(15)      :: X
REAL, DIMENSION(1:5,1:3) :: Y, Z
```

The above are *explicit-shape* arrays.

Terminology:

- **rank** — number of dimensions.

  Rank of `X` is 1; rank of `Y` and `Z` is 2.

- **bounds** — upper and lower limits of indices.

  Bounds of `X` are 1 and 15; Bound of `Y` and `Z` are 1 and 5 and 1 and 3.

- **extent** — number of elements in dimension;

  Extent of `X` is 15; extents of `Y` and `Z` are 5 and 3.

- **size** — total number of elements.

  Size of `X`, `Y` and `Z` is 15.

- **shape** — rank and extents;

  Shape of `X` is 15; shape of `Y` and `Z` is 5,3.

- **conformable** — same shape.

  `Y` and `Z` are conformable.

Declarations

Literals and constants can be used in array declarations,

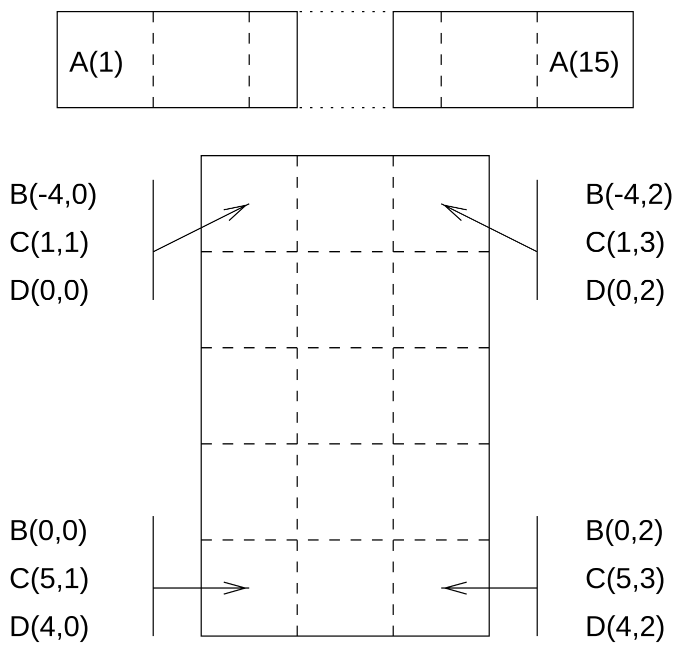
```
REAL, DIMENSION(100)         :: R
REAL, DIMENSION(1:10,1:10) :: S
REAL                         :: T(10,10)
REAL, DIMENSION(-10:-1)      :: X
INTEGER, PARAMETER             :: lda = 5
REAL, DIMENSION(0:lda-1)       :: Y
REAL, DIMENSION(1+lda*lda,10) :: Z
```

- default lower bound is 1,
- bounds can begin and end anywhere,
- arrays can be zero-sized (if `lda = 0`),

Visualisation of Arrays

```
REAL, DIMENSION(15)        :: A
REAL, DIMENSION(-4:0,0:2) :: B
REAL, DIMENSION(5,3)       :: C
REAL, DIMENSION(0:4,0:2)  :: D
```

Individual array elements are denoted by *subscripting* the array name by an `INTEGER`, for example, `A(7)` 7^{th} element of `A`, or `C(3,2)`, 3 elements down, 2 across.



Array Conformance

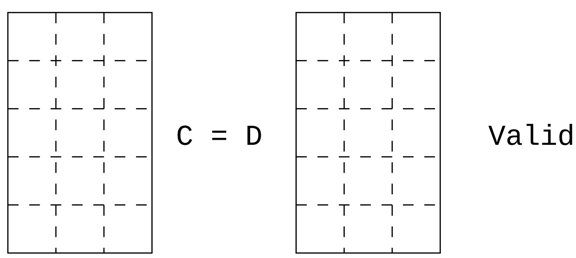
Arrays or sub-arrays must conform with all other objects in an expression:

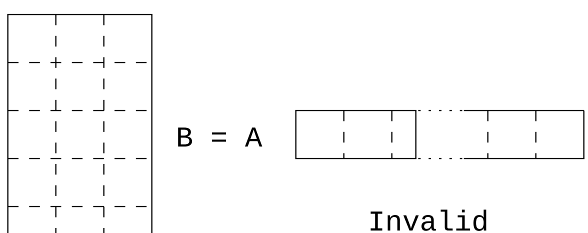
- a scalar conforms to an array of any shape with the same value for every element:

```
C = 1.0   ! is valid
```

- two array references must conform in their shape.

  Using the declarations from before:





`A` and `B` have the same size but have different shapes so cannot be directly equated.

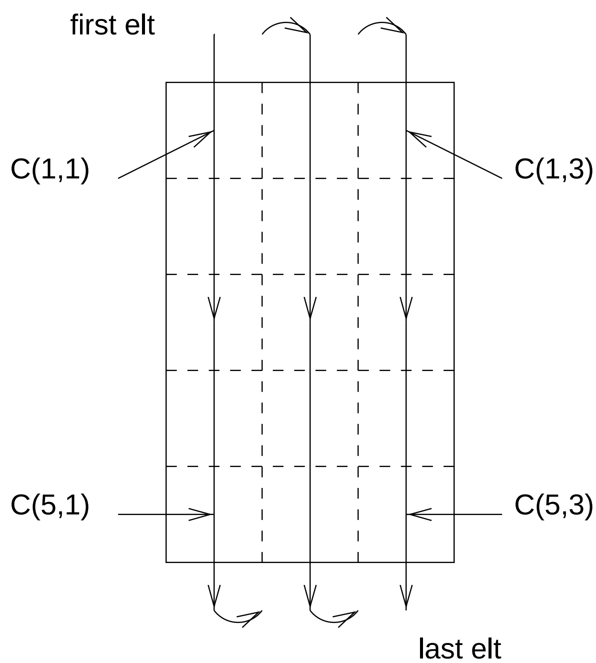
Array Element Ordering

Organisation in memory:

- Fortran 90 does not specify anything about how arrays should be located in memory. **It has no storage association.**

- Fortran 90 does define an array element ordering for certain situations which is of column major form,

The array is conceptually ordered as:




```
C(1,1),C(2,1),..,C(5,1),C(1,2),C(2,2),..,C(5,3)
```

Array Syntax

Can reference:

- whole arrays
  - `A = 0.0`
    sets whole array `A` to zero.
  - `B = C + D`
    adds `C` and `D` then assigns result to `B`.
- elements
  - `A(1) = 0.0`
    sets one element to zero,
  - `B(0,0) = A(3) + C(5,1)`
    sets an element of `B` to the sum of two other elements.
- array sections
  - `A(2:4) = 0.0`
    sets `A(2)`, `A(3)` and `A(4)` to zero,
  - `B(-1:0,1:2) = C(1:2,2:3) + 1.0`
    adds one to the subsection of `C` and assigns to the subsection of `B`.

Whole Array Expressions

Arrays can be treated like a single variable in that:

- can use intrinsic operators between conformable arrays (or sections),

  ```
  B = C * D - B**2
  ```

  this is equivalent to concurrent execution of:

  ```
  B(-4,0) = C(1,1)*D(0,0)-B(-4,0)**2 ! in ||
  B(-3,0) = C(2,1)*D(1,0)-B(-3,0)**2 ! in ||
   ...
  B(-4,1) = C(1,2)*D(0,1)-B(-4,1)**2 ! in ||
   ...
  B(0,2)  = C(5,3)*D(4,2)-B(0,2)**2  ! in ||
  ```

- elemental intrinsic functions can be used,

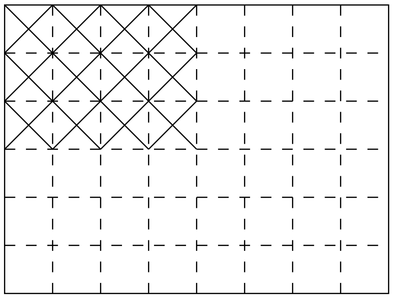
  ```
  B = SIN(C)+COS(D)
  ```

  the function is applied element by element.

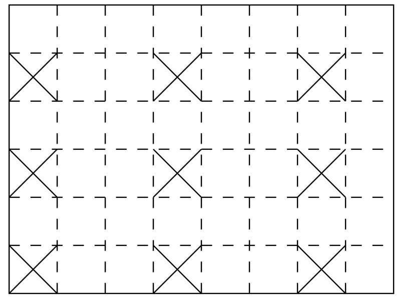
Array Sections — Visualisation

Given,

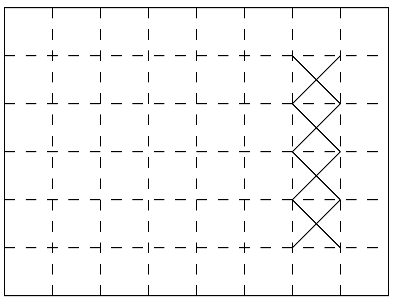
```
REAL, DIMENSION(1:6,1:8) :: P
```



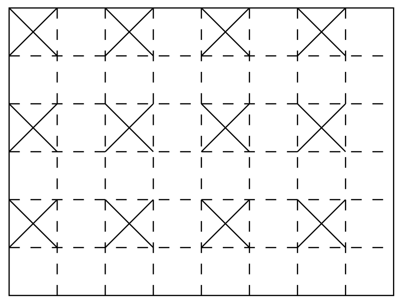
P(1:3,1:4)



P(2:6:2,1:7:3)



P(2:5,7) P(2:5,7:7)



P(1:6:2,1:8:2)

Consider the following assignments,

- `P(1:3,1:4) = P(1:6:2,1:8:2)` and `P(1:3,1:4) = 1.0` are valid.
- `P(2:8:2,1:7:3) = P(1:3,1:4)` and `P(2:6:2,1:7:3) = P(2:5,7)` are not.
- `P(2:5,7)` is a 1D section (scalar in dimension 2) whereas `P(2:5,7:7)` is a 2D section.

Array Sections

subscript-triplets specify sub-arrays. The general form is:

[$<$ *bound1* $>$]:[$<$ *bound2* $>$][:$<$ *stride* $>$]

The section starts at $<$ *bound1* $>$ and ends at or before $<$ *bound2* $>$. $<$ *stride* $>$ is the increment by which the locations are selected.

$<$ *bound1* $>$, $<$ *bound2* $>$ and $<$ *stride* $>$ must all be scalar integer expressions. Thus

```
A(:)         ! the whole array
A(3:9)       ! A(m) to A(n) in steps of 1
A(3:9:1)     ! as above
A(m:n)       ! A(m) to A(n)
A(m:n:k)     ! A(m) to A(n) in steps of k
A(8:3:-1)    ! A(8) to A(3) in steps of -1
A(8:3)       ! A(8) to A(3) step 1 => Zero size
A(m:)        ! from A(m) to default UPB
A(:n)        ! from default LWB to A(n)
A(::2)       ! from default LWB to UPB step 2
A(m:m)       ! 1 element section
A(m)         ! scalar element - not a section
```

are all valid sections.

Array Inquiry Intrinsics

These are often useful in procedures, consider the declaration:

```
REAL, DIMENSION(-10:10,23,14:28) :: A
```

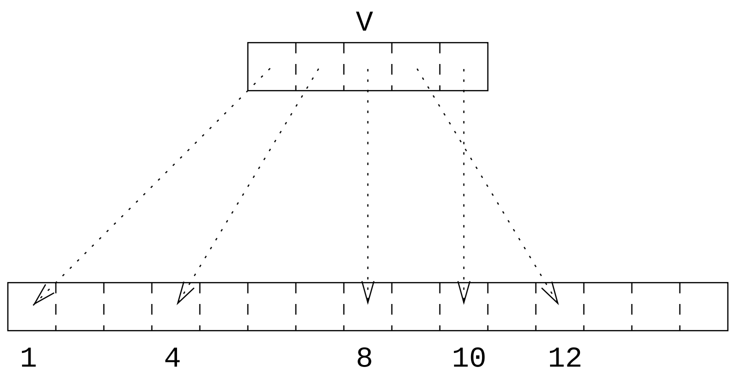
- `LBOUND(SOURCE[,DIM])` — lower bounds of an array (or bound in an optionally specified dimension).
  - `LBOUND(A)` is `(/-10,1,14/)` (array);
  - `LBOUND(A,1)` is -10 (scalar).
- `UBOUND(SOURCE[,DIM])` — upper bounds of an array (or bound in an optionally specified dimension).
- `SHAPE(SOURCE)` — shape of an array,
  - `SHAPE(A)` is `(/21,23,15/)` (array);
  - `SHAPE((/4/))` is `(/1/)` (array).
- `SIZE(SOURCE[,DIM])` — total number of array elements (in an optionally specified dimension),
  - `SIZE(A,1)` is 21;
  - `SIZE(A)` is 7245.
- `ALLOCATED(SOURCE)` — array allocation status;

Vector-valued Subscripts

A 1D array can be used to subscript an array in a dimension. Consider:

```
INTEGER, DIMENSION(5) :: V = (/1,4,8,12,10/)
INTEGER, DIMENSION(3) :: W = (/1,2,2/)
```

- `A(V)` is `A(1)`, `A(4)`, `A(8)`, `A(12)`, and `A(10)`.



- the following are valid assignments:

```
A(V) = 3.5
C(1:3,1) = A(W)
```

- it would be invalid to assign values to `A(W)` as `A(2)` is referred to twice.

- only 1D vector subscripts are allowed, for example,

```
A(1) = SUM(C(V,W))
```

Array Constructors

Used to give arrays or sections of arrays specific values. For example,

```
IMPLICIT NONE
INTEGER                          :: i
INTEGER, DIMENSION(10)           :: ints
CHARACTER(len=5), DIMENSION(3)   :: colours
REAL, DIMENSION(4)               :: heights
heights = (/5.10, 5.6, 4.0, 3.6/)
colours = (/'RED  ','GREEN','BLUE '/)
! note padding so strings are 5 chars
ints    = (/ 100, (i, i=1,8), 100 /)
 ...
```

- constructors and array sections must conform.
- must be 1D.
- for higher rank arrays use `RESHAPE` intrinsic.
- `(i, i=1,8)` is an *implied* `DO` and is 1,2,..,8, it is possible to specify a stride.

The RESHAPE Intrinsic Function

RESHAPE is a general intrinsic function which delivers an array of a specific shape:

```
RESHAPE(SOURCE, SHAPE)
```

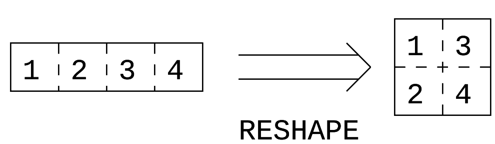
For example,

```
A = RESHAPE((/1,2,3,4/),(/2,2/))
```

A is filled in array element order and looks like:

```
1  3
2  4
```

Visualisation,



Allocatable Arrays

Fortran 90 allows arrays to be created on-the-fly; these are known as *deferred-shape* arrays:

- Declaration:

```
INTEGER, DIMENSION(:), ALLOCATABLE :: ages   ! 1D
REAL, DIMENSION(:,:), ALLOCATABLE :: speed   ! 2D
```

  Note `ALLOCATABLE` attribute and fixed rank.

- Allocation:

```
READ*, isize
ALLOCATE(ages(isize), STAT=ierr)
IF (ierr /= 0) PRINT*, "ages : Allocation failed"

ALLOCATE(speed(0:isize-1,10),STAT=ierr)
IF (ierr /= 0) PRINT*, "speed : Allocation failed"
```

- the optional `STAT=` field reports on the success of the storage request. If the `INTEGER` variable `ierr` is zero the request was successful otherwise it failed.

Deallocating Arrays

Heap storage can be reclaimed using the `DEALLOCATE` statement:

```
IF (ALLOCATED(ages)) DEALLOCATE(ages,STAT=ierr)
```

- it is an error to deallocate an array without the `ALLOCATE` attribute or one that has not been previously allocated space,

- there is an intrinsic function, `ALLOCATED`, which returns a scalar `LOGICAL` values reporting on the status of an array,

- the `STAT=` field is optional but its use is recommended,

- if a procedure containing an allocatable array which does not have the `SAVE` attribute is exited without the array being `DEALLOCATE`d then this storage becomes inaccessible.

Masked Array Assignment — Where Statement

This is achieved using `WHERE`:

```
WHERE (I .NE. 0) A = B/I
```

the LHS of the assignment must be array valued and the mask, (the logical expression,) and the RHS of the assignment must all conform;

For example, if

$$\mathtt{B} = \begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{pmatrix}$$

and,

$$\mathtt{I} = \begin{pmatrix} \boxed{2} & 0 \\ 0 & \boxed{2} \end{pmatrix}$$

then

$$\mathtt{A} = \begin{pmatrix} \boxed{0.5} & \cdot \\ \cdot & \boxed{2.0} \end{pmatrix}$$

Only the indicated elements, corresponding to the non-zero elements of `I`, have been assigned to.

Where Construct

- there is a block form of masked assignment:

```
WHERE(A > 0.0)
 B = LOG(A)
 C = SQRT(A)
ELSEWHERE
 B = 0.0 ! C is NOT changed
ENDWHERE
```

- the mask must conform to the RHS of each assignment; `A`, `B` and `C` must conform;

- `WHERE` ... `END WHERE` is *not* a control construct and cannot currently be nested;

- the execution sequence is as follows: evaluate the mask, execute the `WHERE` block (in full) then execute the `ELSEWHERE` block;

- the separate assignment statements are executed sequentially but the individual elemental assignments within each statement are (conceptually) executed in parallel.

Dummy Array Arguments

There are two main types of dummy array argument:

- *explicit-shape* — all bounds specified;

  ```
  REAL, DIMENSION(8,8), INTENT(IN) :: expl_shape
  ```

  The actual argument that becomes associated with an explicit-shape dummy must conform in size and shape.

- *assumed-shape* — no bounds specified, all inherited from the actual argument;

  ```
  REAL, DIMENSION(:,:), INTENT(IN) :: ass_shape
  ```

  An explicit interface *must* be provided.

- dummy arguments cannot be (unallocated) `ALLOCATABLE` arrays.

Assumed-shape Arrays

Should declare dummy arrays as assumed-shape arrays:

```
PROGRAM Main
 IMPLICIT NONE
  REAL, DIMENSION(40)    :: X
  REAL, DIMENSION(40,40) :: Y
   ...
  CALL gimlet(X,Y)
  CALL gimlet(X(1:39:2),Y(2:4,4:4))
  CALL gimlet(X(1:39:2),Y(2:4,4)) ! invalid
CONTAINS
 SUBROUTINE gimlet(a,b)
  REAL, INTENT(IN)   :: a(:), b(:,:)
    ...
 END SUBROUTINE gimlet
END PROGRAM
```

Note:

- the actual arguments cannot be a vector subscripted array,
- the actual argument cannot be an assumed-size array.
- in the procedure, bounds begin at 1.

Automatic Arrays

Other arrays can depend on dummy arguments, these are called *automatic* arrays and:

- their size is determined by dummy arguments,
- they cannot have the `SAVE` attribute (or be initialised);

Consider,

```
PROGRAM Main
 IMPLICIT NONE
  INTEGER :: IX, IY
 .....
  CALL une_bus_riot(IX,2,3)
  CALL une_bus_riot(IY,7,2)
CONTAINS
 SUBROUTINE une_bus_riot(A,M,N)
  INTEGER, INTENT(IN) :: M, N
  INTEGER, INTENT(INOUT) :: A(:,:)
  REAL :: A1(M,N)                 ! auto
  REAL :: A2(SIZE(A,1),SIZE(A,2)) ! auto
   ...
 END SUBROUTINE
END PROGRAM
```

The `SIZE` intrinsic or dummy arguments can be used to declare automatic arrays. `A1` and `A2` may have different sizes for different calls.

Random Number Intrinsic

- `RANDOM_NUMBER(HARVEST)` will return a scalar (or array of) pseudorandom number(s) in the range $0 \leq x < 1$.

  For example,

  ```
  REAL                     :: HARVEST
  REAL, DIMENSION(10,10) :: HARVEYS
  CALL RANDOM_NUMBER(HARVEST)
  CALL RANDOM_NUMBER(HARVEYS)
  ```

- `RANDOM_SEED([SIZE=`< *int* >`])` finds the size of the seed.

- `RANDOM_SEED([PUT=`<*array*>`])` seeds the random number generator.

```
CALL RANDOM_SEED(SIZE=isze)
CALL RANDOM_SEED(PUT=IArr(1:isze))
```

Vector and Matrix Multiply Intrinsics

There are two types of intrinsic matrix multiplication:

- DOT_PRODUCT(VEC1, VEC2) — inner (dot) product of two rank 1 arrays.

  For example,

  ```
  DP = DOT_PRODUCT(A,B)
  ```

  is equivalent to:

  ```
  DP = A(1)*B(1) + A(2)*B(2) + ...
  ```

  For LOGICAL arrays, the corresponding operation is a logical .AND..

  ```
  DP = LA(1) .AND. LB(1) .OR. &
       LA(2) .AND. LB(2) .OR. ...
  ```

- MATMUL(MAT1, MAT2) — 'traditional' matrix-matrix multiplication:

  - if MAT1 has shape (n, m) and MAT2 shape (m, k) then the result has shape (n, k);
  - if MAT1 has shape (m) and MAT2 shape (m, k) then the result has shape (k);
  - if MAT1 has shape (n, m) and MAT2 shape (m) then the result has shape (n);

  For LOGICAL arrays, the corresponding operation is a logical .AND..

Array Location Intrinsics

There are two intrinsics in this class:

- `MINLOC(SOURCE[,MASK])`— Location of a minimum value in an array under an optional mask.
- `MAXLOC(SOURCE[,MASK])`— Location of a maximum value in an array under an optional mask.

A 1D example,

`MAXLOC(X) = (/6/)`

7	9	-2	4	8	10	2	7	10	2	1

A 2D example. If

$$\texttt{Array} = \begin{pmatrix} 0 & -1 & 1 & 6 & -4 \\ 1 & -2 & 5 & 4 & -3 \\ 3 & 8 & 3 & -7 & 0 \end{pmatrix}$$

then

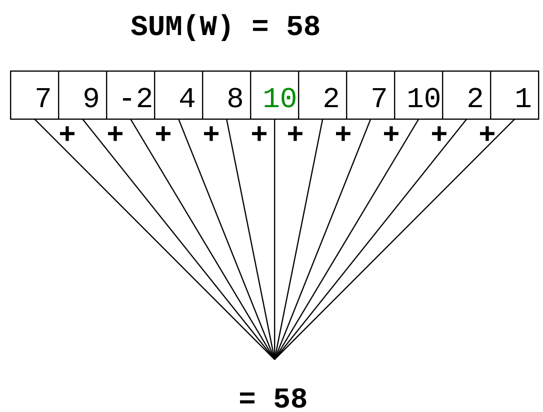
- `MINLOC(Array)` is `(/3,4/)`
- `MAXLOC(Array,Array.LE.7)` is `(/1,4/)`
- `MAXLOC(MAXLOC(Array,Array.LE.7))` is `(/2/)` (array valued).

Array Reduction Intrinsics

- `PRODUCT(SOURCE[,DIM][,MASK])`— product of array elements (in an optionally specified dimension under an optional mask);

- `SUM(SOURCE[,DIM][,MASK])`— sum of array elements (in an optionally specified dimension under an optional mask).

The following 1D example demonstrates how the 11 values are reduced to just one by the `SUM` reduction:



Consider this 2D example, if

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

- `PRODUCT(A)` is 720

- `PRODUCT(A,DIM=1)` is (/2, 12, 30/)

- `PRODUCT(A,DIM=2)` is (/15, 48/)

Array Reduction Intrinsics (Cont'd)

These functions operate on arrays and produce a result with less dimensions that the source object:

- ☐ `ALL(MASK[,DIM])`— `.TRUE.` if *all* values are `.TRUE.`, (in an optionally specified dimension);
- ☐ `ANY(MASK[,DIM])`— `.TRUE.` if *any* values are `.TRUE.`, (in an optionally specified dimension);
- ☐ `COUNT(MASK[,DIM])`— number of `.TRUE.` elements in an array, (in an optionally specified dimension);
- ☐ `MAXVAL(SOURCE[,DIM][,MASK])`— maximum Value in an array (in an optionally specified dimension under an optional mask);
- ☐ `MINVAL(SOURCE[,DIM][,MASK])`— minimum value in an array (in an optionally specified dimension under an optional mask);

If `DIM` is absent or the source array is of rank 1 then the result is scalar, otherwise the result is of rank $n-1$.